2. Memory allocator

Master M1 MOSIG, Grenoble Universities
Arnaud Legrand, Benjamin Negrevergne

2010

Abstract

This assignment is about implementing and evaluating the performance of classical dynamic memory allocators. To this end, we provide you a tarball comprising a C skeleton, a Makefile and some non-regression tests. The first part is devoted to implementing the classical first fit memory allocator (using a list of free blocs sorted by address) in the C skeleton with the help of the tests provided in the Makefile. Passing these tests is considered as mandatory.

Then, you should be able to easily implement other allocators like best fit or worst fit, and test them in similar situations (we only provide test cases for the first fit allocation).

Finally, you should try to evaluate the performance of your allocators using simple (but real) programs of your choice. The aim of this study is to check whether the relative performance of classical heuristics follows indeed what was announced during the lecture.

You should work by groups of three people and turn in both your (clean and working !!!) code and a short design document (no more than 10 pages) explaining your choices and the results of your study.

1 Implementing a dynamic memory allocator

I. Allocating memory

We propose to study a system where a fixed amount of memory is allocated at the initialization. This fixed amount of memory (a static array of `char`s will be the only space available for dynamic allocation. The challenge is to provide a mechanism to manage this memory. Managing the memory means :

- to know where the free memory blocs are;
- to slice and allocate the memory for the user when it is needed;
- to free the memory blocs when the user does not need them anymore.

II. Linking free blocs, and available blocs

A memory allocator algorithm is based on the principle of linking free memory blocs. Each free bloc is associated with a descriptor that contains its size and a pointer to the next free bloc. **This**

descriptor must be placed inside the managed memory itself. The principle of the algorithm is the following :
When one need to allocate `bloc_size` bytes, the allocator must go through to lists of free blocs and find a free bloc that is big enough. Let `b` be one of these blocs. It may be chosen according to the following criteria :

- **First big enough free bloc (first fit) :** We choose the first bloc `b` so that `size(b) >= bloc_size`. This policy aims at having the fastest research.
- **Smallest waste (best fit) :** We choose the bloc `b` that has the smallest waste. In other words we choose the bloc `b` so that `size(b) - bloc_size` is as small as possible.
- **Biggest waste (worst fit) :** We choose the bloc so that `size(b) - bloc_size` is as big as possible.

III. Implementing the allocator and validating your implementation

We want you to create a memory allocator. The "memory" that is to be managed is declared as an array of chars :
```
#define MEMORY_SIZE 512
char memory[MEMORY_SIZE]
```
A pointer to this memory will be passed to the initializing function `memory_init()`. At the beginning, the list of free blocs will be made of a single big free bloc. The free bloc descriptor placed at the beginning of a free bloc can be declared like this :

```
struct free_bloc\{
  int size
  struct free_bloc *next;
  /* ... */
};
```

Question III.1: *Do you have to keep a list for occupied blocs ? If no, explain why, if yes, provide a* `C` *definition of the structure.*

The allocator must implement the following methods :
`void memory_init(char *mem, int size);`
Initialize the list of free blocs with a single free bloc corresponding to the full array.
`void *memory_alloc(int size);`
This method allocates a bloc of size `size`. It returns a pointer to the allocated memory or `NULL` if the allocation failed.
`void memory_free(void *zone);`
This method receives an address to an occupied bloc. It updates the list of the free blocs and merge contiguous blocs.

Question III.2: *As a user, can you use addresses 0,1,2 ? Generaly speeking, can a user manipulate any addresses ?*

Question III.3: *When a bloc is allocated, what address should be given to the user ?*

Question III.4: *When a bloc is allocated inside a free memory zone, one must take care of how the memory is partitioned. In the most simple case, we allocate the beginning of the bloc for our needs, and the rest of it becomes a new free bloc. However it is possible that the rest is too small. How is it possible ? How would you deal with this case?*

Retrieve and extract the archive at :
`mandelbrot:~negreveb/ens/2010/mosig/memory_alloc.tgz`
The archive contains :

- `mem_alloc.c` : A stub for your memory allocator library.
- `Makefile` : A Makefile to build and test your memory allocator.
- `README.tests` : A file containing a short description for each provided test.

2 Measuring Fragmentation

Now that you have implemented a few strategies (among which should be at least first fit with a list of free blocs sorted by address, best fit, and worst fit), it is time to evaluate these strategies. As we already have explained during the lecture, the evaluation of these strategies should be done using sound workload [WJNB95]. This is why the Makefile provided in the tarball provides you with the mechanism enabling to run program with your own version of malloc/free/realloc instead of the ones provided by the libc. Hence, you will be able to run more complex programs than the synthetic ones you used in the previous part to check the correctness of your implementation.

Yet, before starting any evaluation, we will need a way to measure fragmentation. A few metrics have been proposed in the literature [JW98]. We reproduce a part the work of Johnstone and Wilson explaining possible approaches:

> *There are a number of legitimate ways to measure fragmentation. Figure 1 illustrates four of these. Figure 1 is a trace of the memory usage of the GCC compiler using a given allocator. The lower line is the amount of live memory requested by GCC (in kilobytes). The upper line is the amount of memory actually used by the allocator to satisfy GCC's memory requests.*
>
> *The four ways to measure fragmentation for a program which we considered are:*
>
> 1. *The amount of memory used by the allocator relative to the amount of memory requested by the program, averaged across all points in time. In Figure 1, this is equivalent to averaging the fragmentation (i.e. the ratio) for each corresponding point on the upper and lower lines for the entire run of the program. For the program of Figure 1,this measure yields 258% of fragmentation. The problem with this measure of fragmentation is that it tends to hide the spikes in memory usage, and it is at these spikes where fragmentation is most likely to be a problem.*

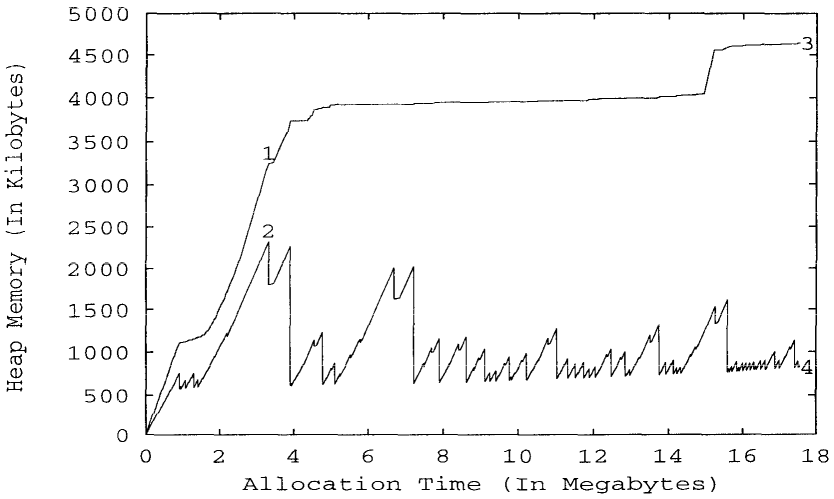

Figure 1: Measuring fragmentation

2. *The amount of memory used by the allocator relative to the maximum amount of memory requested by the program at the point of maximum live memory. In Figure 1 this corresponds to the amount of memory at point 1 relative to the amount of memory at point 2. In this example, this measure yields 39.8% fragmentation. The problem with this measure of fragmentation is that the point of maximum live memory is usually not the most important point in the run of a program. The most important point is likely to be a point where the allocator must request more memory from the operating system.*
3. *The maximum amount of memory used by the allocator relative to the amount of memory requested by the program at the point of maximal memory usage. In Figure 1 this corresponds to the amount of memory at point 3 relative to the amount of memory at point 4. On this example, this measure yields 462% fragmentation. The problem with this measure of fragmentation is that it will tend to report high fragmentation for programs that use only slightly more memory than they request if the extra memory is used at a point where only a minimal amount of memory is live.*
4. *The maximum amount of memory used by the allocator relative to the maximum amount of live memory. These two points do not necessarily occur at the same point in the run of the program. In Figure 1 this cor- responds to the amount of memory at point 3 relative to the amount of memory at point 2. On this example, this measure yields 100% fragmentation. The problem with this measure of fragmentation is that it can yield a number that is too low if the point of maximal memory usage is a point with a small amount of live memory and is also the point where the amount of memory used becomes problematic.*

We obviously do not expect from you a study as thorough and deep as the one provided by Johnstone and Wilson. Yet, you should try to evaluate the performance of a few real programs (say sort, head, ls, dmesg, ...) with one (or more) of the previous metric or even one of your own invention. This should enable you to check whether "Best-Fit $\approx$ First Fit $\gg$ Worst fit" or not.

References

[JW98] Mark S. Johnstone and Paul R. Wilson. The memory fragmentation problem: Solved. In *Proceedings of the First International Symposium on Memory Management, ACM*. Press, 1998. Available at `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.141.4610`.

[WJNB95] Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles. Dynamic storage allocation: A survey and critical review. pages 1–116. Springer-Verlag, 1995. Available at `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.111.8237`.